Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations

Vandi Verma[1]
QSS Inc., NASA Ames Research Center, Moffett Field, CA, 94035

Ari Jónsson[2]
USRA-RIACS, NASA Ames Research Center, Moffett Field, CA, 94035

Corina Pasareanu[3]
QSS Inc., NASA Ames Research Center, Moffett Field, CA, 94035

and

Michael Iatauro[4]
QSS Inc., NASA Ames Research Center, Moffett Field, CA, 94035

[Abstract] This paper presents an execution engine and the associated execution language for spacecraft operations. The software tool highlighted is the Universal Executive execution system and the language is PLEXI. PLEXIL is a lightweight, well-defined, predictable, and verifiable language capable of expressing spacecraft control concepts used by human operators and many high-level automated planners. The Universal Executive is a lightweight execution system that can autonomously execute PLEXIL plans on spacecraft including satellites, robots, instruments, and habitats. The Universal Executive and the PLEXIL language allow variable levels of autonomy and co-ordination with various other spacecraft systems, and human operators.

I. Introduction

Fault-tolerant command sequence (plan) execution and monitoring is a cornerstone of spacecraft operations. This is true irrespective of whether the plans to be executed are generated autonomously by an internal closed-loop control process on the spacecraft or by external ground-based planners. In practice many spacecraft combine internal decisions and externally generated commands (plans); for example externally-defined command sequences interact with internal fault protection decisions. Existing execution engines (executives) vary greatly between spacecraft; their capabilities are highly specialized to the needs of the mission in which they are employed. Custom implementations have made the reuse of execution and planning frameworks more difficult, and has all but precluded information sharing between different execution and decision making systems.

To help address these issues, we have developed a general plan execution language, called the Plan Execution Interchange Language (PLEXIL), along with the Universal Executive (UE), a general execution engine that executes PLEXIL. As general execution tools, PLEXIL and the UE are intended for use on a variety of spacecraft, robots, instruments, and habitats. This simplifies operations, unifies interfaces to operators and other systems, and results in a robust execution system that improves with every flight. However, general frameworks often tend to be monolithic and slow. Care was taken to avoid this.

[1] Research Scientist, Code TI, Autonomy + Robotics and Discover and Systems Health Management, MS 269-2 NASA Ames Research Center Moffett Field CA 94035, AIAA Member.
[2] Senior Staff Scientist, Autonomy and Robotics, MS 269-2 NASA Ames Research Center Moffett Field CA 94035, n/a.
[3] Research Scientist, Robust Software Engineering, M/S 269-2 Moffett Field, CA 94035, n/a.
[4] Intern, Code TI, Autonomy and Robotics, MS 269-2 NASA Ames Research Center Moffett Field CA 94035, n/a.

Most command sequencing languages and execution engines used on spacecraft are highly simplified to ensure that spacecraft safety can be guaranteed under all execution paths. This has ruled out systems that allow complex logic. Although the PLEXIL language is structurally simple, it is capable of expressing complex control constructs that include conditionals, branches, floating contingencies, loops, event-driven control, time-based control, etc. The language has well defined semantics, including guarantees of unambiguous responses for any given plan and situation. The Universal Executive is lightweight and implements the exact semantics specified by the language and is designed to be reusable as the core executive in a variety of applications involving different underlying control software and different interactions with decision-making capabilities.

Complex systems must be operated in different ways at different times or in different contexts. For example, spacecraft operations vary according to mission phase. Furthermore, operations may fail and equipment may break or behave off-nominally. Control plans must specify what to do in all anticipated situations including how to respond to failures and problems.

Multiple solutions exist for the execution problem. Programming languages provide one extreme option; a program can be encoded to execute and implement a control strategy. However, this approach brings into play the full complexity of programming languages, which makes development and validation more difficult. In addition, many programming languages require compilation and software installation to be done before a program can be executed on a given system. The other extreme is table-driven or parameter-driven software. In this scenario, the software is pre-defined, but certain aspects of how it operates are defined by parameters or tables. For example, a temperature controller may have a simple program whose parameters are the desired temperature and permitted deviations.

In between those two extremes are languages that express control strategies or plans. Some such languages are akin to common programming languages, thus providing the full power of programming, but also providing some structure to express plans, conditions, steps, actions, etc. Other languages are designed to only express plans that by appropriate execution software. The advantages of languages that are not tied to programming languages is that their expressiveness can be circumscribed and their semantics clearly defined. These properties are crucial for validation, checking, and testing of plans. However, few existing languages have well-defined semantics and many are not powerful enough to express complex control plans.

In developing PLEXIL and the UE, our objective is to provide a well-defined language for expressing plans that control complex systems. PLEXIL is an expressive language that can represent complex plans with loops, conditions, contingencies and other necessary control structures, while also having clear formal semantics that make the expected outcome of execution unambiguous and enable plan validation. Accompanying this language is an example execution engine, the Universal Executive, designed to implement the PLEXIL language and provide interfaces to the controlled system.

In section II we describe example usage scenarios for PLEXIL and the UE. Section III provides an overview of PLEXIL syntax, and section IV is an overview of language semantics. A summary of verification and validation tools that have been applied to PLEXIL is presented in section V. Some features of the UE implementation are discussed in section VI. Demonstration scenarios and ongoing application development is discussed in section VII.

II. Usage Scenarios

The tools presented in this paper offer a number of distinct advantages over current command load representations and command execution engines. We describe a few scenarios to illustrate their benefit.

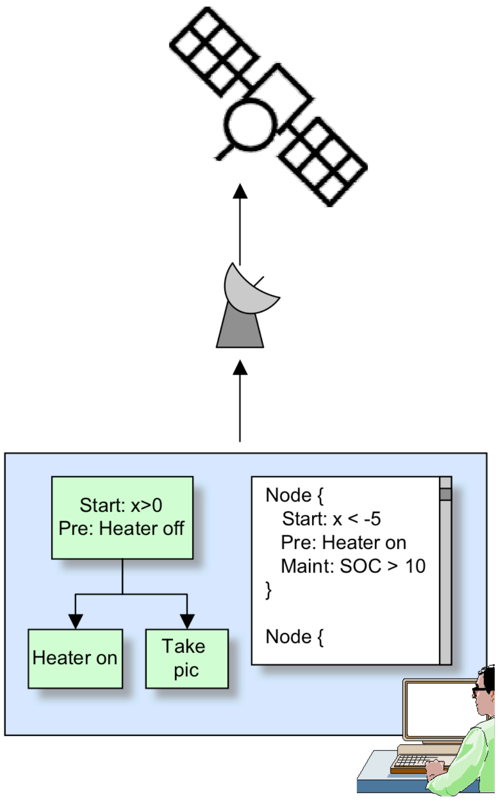

Figure 1. Manually crated PLEXIL plans.

A. Manually generated command upload and execution

This scenario is similar to a large proportion of current deep space spacecraft and Mars rover operations. Figure 1 provides an illustration. A command load is manually written in a scripting language with or without the use of graphical user interfaces and simulators. It is then reviewed and tested by a committee of experts, revised if necessary, and when it is cleared by the experts, it is uploaded to the spacecraft and executed. PLEXIL and the Universal Executive can easily be used in any of these scenarios.

PLEXIL is designed to express and enable context-sensitive execution, not possible in typical command sequences and scripts. This is a useful feature in scenarios where command upload has a lot of latency and there is uncertainty about the situations the spacecraft may encounter. PLEXIL also represents complex logic in human-readable formats unlike in many examples. Finally, PLEXIL is designed with verification and validation in mind. A Universal Executive running on-board the spacecraft executes plans based on PLEXIL semantics. These semantics have been formally verified for correctness and predictable execution.

B. Automatically generated external plans

In this scenario, planning is done externally by an automated decision making system. When a planner is ground-based, it can use extensive knowledge bases and high fidelity models for planning since ground-data systems are far less computationally constrained than spacecraft processors. When plans are translated into PLEXIL the state space is much smaller than the space of possible plans that a general planner can create, and can thus fit within the scope of existing verification and validation tools. Tools designed for PLEXIL can check these plans for syntactic and semantic errors, including deadlocks, race condition, reference to undefined commands or sensor values.

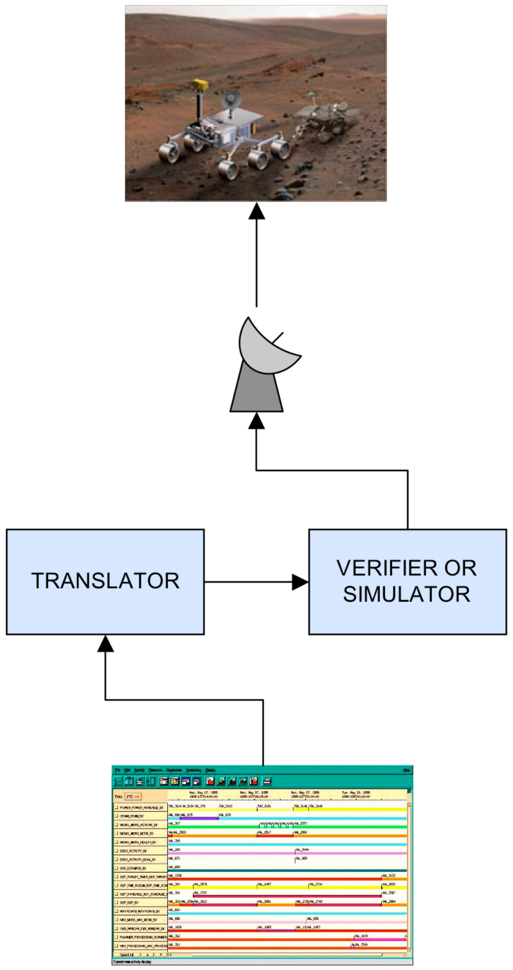

Figure 2. Automatically created external plans.

C. Plans created in-situ

In this scenario, decision making is done on-board the spacecraft and there is a feedback loop between the automated decision-making system and executive. PLEXIL plans do not have to be complied. The language has an efficient interface for continuous planning that allows plans to be updated dynamically without violating any of the guarantees provided by PLEXIL plans. In addition PLEXIL has interfaces to continuously update decision making systems about the status of execution, and current state of the spacecraft.

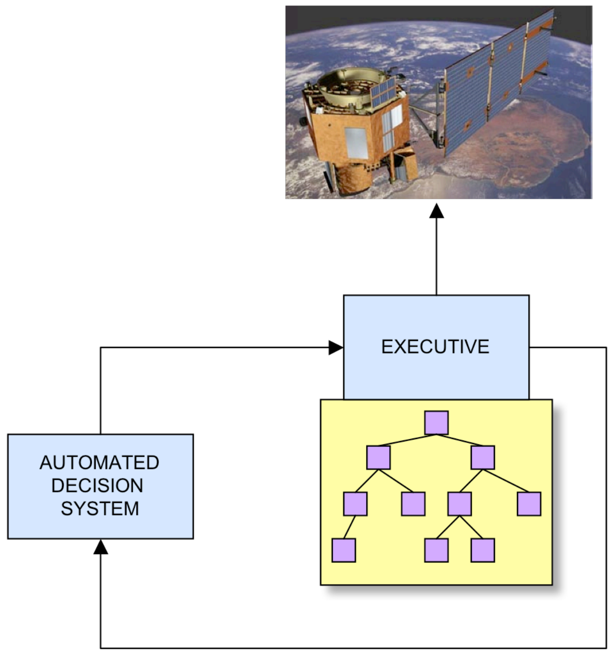

Figure 3. Plans created in-situ.

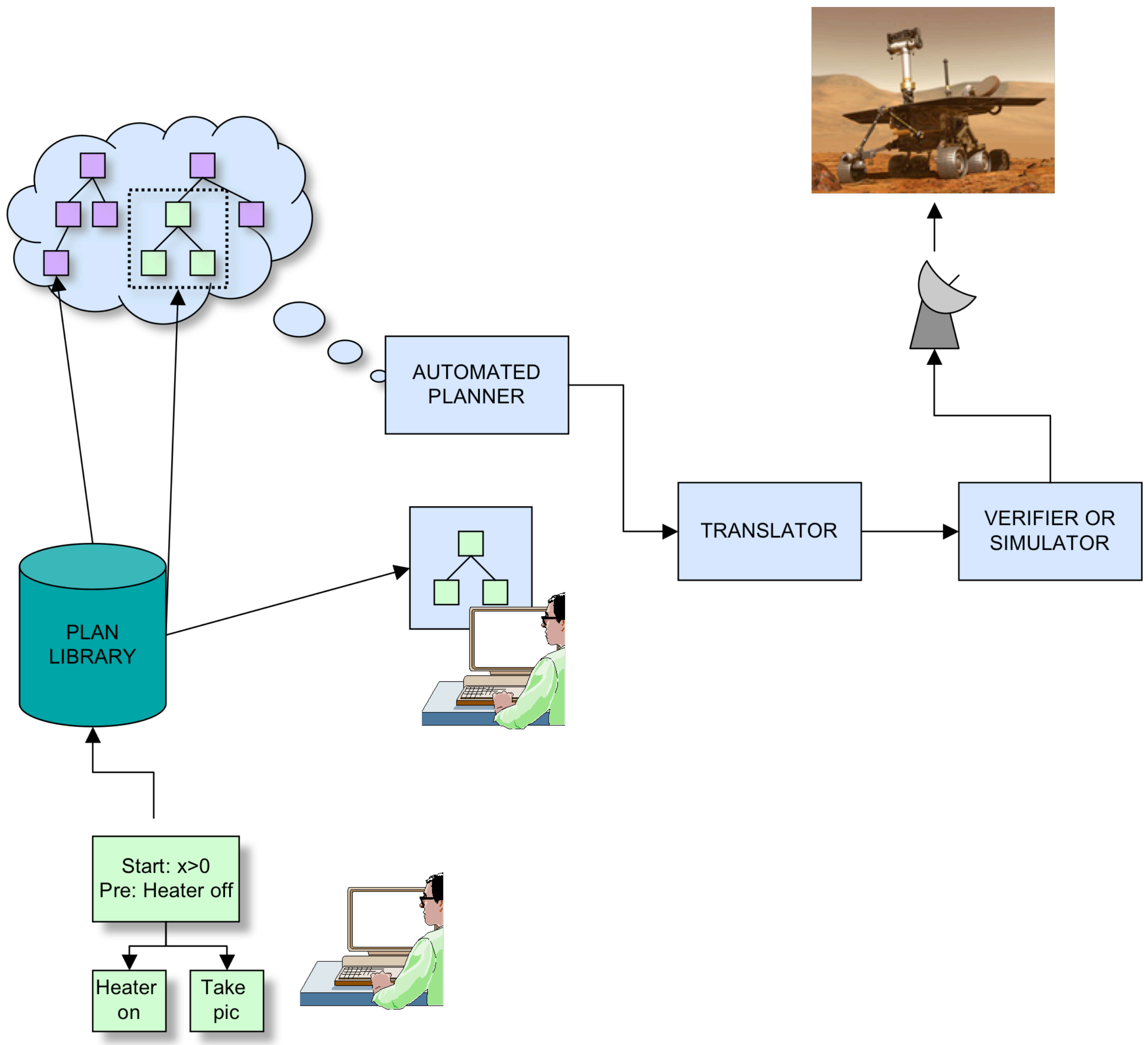

Figure 4. Mixed manual and automated planning.

D. Plans created in mixed manual and automated mode

In this scenario some modular sub-plans for well known situations are created beforehand. Ground and flight controllers may create additional sub-plans during the mission. During mission execution these sub-plans are used by an external or in-situ automated planner to assemble more complex plans. In PLEXIL the behavior and interface of the sub-plans in the library of sub-plans is well-defined. PLEXIL interfaces allow easy integration of sub-plans into more complex plans.

III. PLEXIL Syntax

Plan Execution Interchange Language (PLEXIL) is a small, well-defined language that provides a great deal of power in terms of describing complex plans, while being easy to formally validate and verify. This feature has significant implications in terms of representing and reasoning about control plans. For a long time, the specification of spacecraft operation plans like command sequences has been limited to very simple control structures. This has largely been due to the difficulty of specifying control structures like conditions and contingencies in such plans, without making the expected semantics unclear and the validation problem unsolvable. Existing languages with expressive representations do so at the cost of formal guarantees about execution.

Another novel feature is the approach used to access information from the underlying functional layer, which avoids the requirement that there be a telemetry database, and supports different ways of accessing state information. The *lookup* feature and the ability to perform internal storage and calculations make it possible to replicate most any behavior of database-driven systems without requiring a specific data base. Not requiring a database simplifies the implementation of an executive for PLEXIL and can in many cases make the executives significantly faster.

A. Overview

The fundamental building block of the PLEXIL language is a *node*. A node has two primary functional components, a set of conditions that drive the execution of that node, and the "content" of the node, which specifies what gets executed. The nodes are arranged in a hierarchical fashion, providing controllability at different levels of abstractions. Executing a high level node means enabling the evaluation of the next lower level set of nodes. In terms of content, there are two primary kinds of nodes.

1) List nodes specify hierarchical relationships, each specifying its "child" nodes in the hierarchy. Execution of child nodes is driven by the execution of parent nodes; e.g., a parent node executing enables the evaluation of child nodes.
2) Action nodes are at the leaves of the hierarchy and they perform some actions. These actions include commanding systems, performing calculations, providing updates, and submitting information to other components.

The conditions for each node drive the execution of the node. There are nominal control conditions that specify when the node should start executing, when it should finish executing, and when it should be repeated:

1) Start conditions specify when the node should start execution
2) End conditions specify when the node should finish its execution
3) Repeat conditions specify when the node should be repeated, i.e., made eligible for a repeat execution

Then there are failure conditions that identify when node execution is not successful:

1) Pre-conditions are checked after start conditions become true and verify that it is safe to execute the node. If the check fails, the node cannot be safely executed and will be aborted and marked as having failed.
2) Post-conditions are checked after the node has completed execution and verify that the node did what it was supposed to. If the check fails, the node did not execute correctly and will be marked as having failed.
3) Invariant conditions are checked during node execution and monitor any conditions that are needed for the safe continued execution of the node. If the check fails at any point during execution, the node execution cannot be safely continued, the node will be aborted and marked as having failed.

The simple combination of these conditions and hierarchical nodes provides a great deal of power with very simple linguistic structure. Local variable declarations (of type boolean, numeric, string, and time) and operations on these variables are also supported. Conditions, contingencies, and loops can be implemented, which provides virtually all desired control capabilities for plans, procedures and other types of operations control specifications. At the same time, the very simple language makes it possible to define formal semantics and develop methods for validation of instances.

The core language has almost no redundancy and hence it is small, but this can make it arduous to specify plans using only those simple constructs. To alleviate this and to make the language more usable, syntactic short cuts are used. Example of short cuts include if-then-else statements, for loops, while loops, macros, etc. Any such short cut maps directly into instances of the core language, so no semantics need to be added and core validation techniques can be applied.

The PLEXIL language has a great deal of flexibility in terms of interfacing with the underlying controlled system. There are two primary elements to this interface; one is the commanding interface and the other is the sensing interface. The commanding interface is relatively straightforward but it supports a range of different behaviors for the command execution. The simplest approach is a command that gets executed without any further feedback, except through sensed effects. The next level is a command that provides a return value; this can either be a confirmation of execution being initiated or the result of the outcome. The latter means supporting blocking commands. Finally, some commands provide an initial return (or not) and then send a result after the command execution process is completed. Each of these is supported by PLEXIL, but what is particularly notable about PLEXIL in this context is the ability to have arbitrary logical combination of sensor inputs provide indications of the outcome. Some aspects of the PLEXIL syntax are described in further detail below:

B. Sensing interface

The syntax used for sensing world states in PLEXIL is called a *Lookup*. It is a read-only interface that "looks up" values of world states, or *measurements*. Each external sensor value or world state that might be accessed via a lookup is identified by a domain-specific *measurement* name, e.g., "Temperature". Lookups can appear in *Assignments* or *Conditions*. PLEXIL provides three types of Lookups :

LookupOnChange:

`LookupOnChange("Temperature", 2)` is an event-based repeated lookup that returns an initial value immediately and then repeatedly returns the state value whenever it changes. A "minimal change" parameter may be specified to restrict the value to be returned only when it changes by more than the specified tolerance, which is 2 (ºC) in the example above.

LookupWithFrequency:

`LookupWithFrequency("Temperature", 10)` is a repeated lookup. The initial value is returned immediately. The second parameter specifies a frequency for checking subsequent state value. In this case it specifies that the value of state "Temperature" should be checked 10 times per second.

LookupNow:

`LookupNow("Temperature"}` returns the current value of state "Temperature" immediately.

PLEXIL plans can be written in PLEXIL with syntactic enhancements, or they can be written in core PLEXIL using the context free grammar, or they can be written in XML using the PLEXIL XML schema definition. In addition a graphical editor for PLEXIL plans is also available.

IV. PLEXIL Execution Semantics

A. Overview

One of the main features of PLEXIL is that it has well defined semantics of execution. Clear semantics make it easier to do verification and validation and to implement a light-weight execution that conforms to the semantics of execution. The execution of PLEXIL plans is predictable. In other words, given the same sequence of measurements the execution is *deterministic*. The semantics and key properties of the language, such as determinism, are described formally in (Ref. 1). A PLEXIL plan consists of a set of nodes with hierarchical, temporal, and constraint relationships.

Node Types:

- *Command node*
- *Assignment node*
- *NodeList node*
- *FunctionCall node*
- *Update node*
- *PlanRequest node*

Command, Assignment, FunctionCall, Update, and PlanRequest nodes are action nodes (i.e. leaves) and NodeList nodes are internal nodes. Command nodes issue commands that drive the system being operated on. Assignment nodes perform local computation. FunctionCall nodes access external functions that perform computations, but do not (directly) alter the state of the system. Update nodes provide information to the planning and decision-support interface. PlanRequest nodes request a new execution plan. List nodes may have one or more child nodes. They are containers that provide scope for child nodes, but do not perform any explicit action. We first describe the execution of individual nodes and then we describe the execution of node trees.

B. Node Execution

The semantics of node execution is specified in terms of node states and transitions between node states that are triggered by condition changes.

Node States:

Each node must be in one and only one of the following states at any given time:

- *Inactive*
- *Waiting*

- *Executing*
- *Finishing*
- *Iteration_Ended*
- *Failing*
- *Finished*

Node Transitions:

The set of condition changes that cause node state transitions are as follows:

- *StartCondition T*
- *InvariantCondition F*
- *EndCondition T*
- *Ancestor_inv_condition F*
- *Ancestor_end_condition T*
- *All_children_waiting_or_finished T*
- *Command_abort_complete T*
- *Function_abort_complete T*
- *Parent_executing T*

Nominal Execution:

All the nodes are initialized to state Inactive, except one node (representing the tree root) which is initialized to state Waiting. An Inactive node does not affect the external system at all. When the parent of a node enters state Executing the node is activated, by transitioning it to state Waiting. A node remains in state Waiting until its start condition is met. The default start condition is True which implies that the node may execute immediately upon activation. In state Executing a node or its children perform the main body of execution actions. Upon completion of action (e.g. command or assignment finished) leaf nodes transition to state Iteration_Ended from which they can transition either back to state Waiting (for looping nodes) or to state Finished, which signals the end of execution for the node, including all loops for repeating nodes. The execution of list nodes proceeds similarly, except that there is an extra state Finishing, which implies that the execution purpose of the node is complete and the node is only waiting for running child nodes to finish. If an ancestor node loops the node may once again come into existence, but not otherwise.

Execution in the Presence of Failures:

When a failure occurs (i.e. one of the pre-, post- or maintainance conditions fail) a list node enters state Failing and causes the sub-tree to abort in a deterministic manner. Only in the case of a failure does a parent node abort a child node. State Iteration_Ended is only for looping nodes. It signals the end of a single iteration of execution.

Node Termination:

There are three main causes for node termination: completion of execution, external events, and faults. The default completion of a node depends on the type of node. Assignment and command nodes end when the assignment is complete or function call returns. A hierarchical node ends when all its child nodes have finished. External events or cascading effects of external events may satisfy the explicit end condition of a node. When the end condition of a list node is satisfied and some of its child nodes are still executing the node cleanly terminates its child nodes. PLEXIL semantics for clean termination allow running “clean-up” nodes to ensure safe termination of processes.

The semantics of clean termination of a list node with running children are :

- Only currently executing and just-activated child nodes continue to run
- Pending child nodes whose start conditions are not satisfied are not run
- Parent node waits for active child nodes to finish executing

Faults can also drive node termination. A node fails if its invariant conditions are violated or pre or post conditions are not satisfied. When a node faults it aborts its child nodes without clean-up. When a node fails no more events are processed by the sub-tree rooted at that node. All clean-up actions are handled by sibling nodes.

During execution, the outcome of a node is set to record the current execution status. The outcome is a node attribute that provides additional information about the result of node execution. A node may have any one of the following outcomes:

- *Success*
- *Failure*
- *Skipped*
- *Unknown*

The node outcome is initialized to Unknown. The outcome is set to Skipped if the node did not run, and to Success if the current iteration completed successfully. The outcome is set to Failure if a failure happened. Outcomes provide information only for the current iteration; they are reset for repeating nodes.

Note that all conditions are checked once upon transition to a state in which they apply. Also note that a start condition changing to false is not a condition change. Once a condition is enabled it stays enabled until it is explicitly *reset*. The conditions are only reset for repeating nodes.

(Appendix A : Node state transition diagrams) provides the complete set of node state transitions that govern the semantics of the execution of a single node. In certain states, e.g. state Waiting, all node types have the same semantics. In other cases, such as state Executing, the semantics depend on the node type (list, command, assignment…). For efficiency we represent ancestor end conditions. These are easily computed from the immediate parent and child nodes. In principle, a node only needs to know about its single immediate parent and all child nodes.

PLEXIL plans are modular. PLEXIL allows safe execution of modular sub-plans (plans from a library) in the nominal case, including allowing the sub-plans authority over when execution of the sub-plan is complete. The only restriction is that child nodes may not loop if the parent is complete, but they may run to completion. One of the features of PLEXIL is that it is very expressive. It is simple to represent command sequences where there is a pre-defined ordering, but it is just as simple to represent a large number of monitors that may need to be executed at any instance, or in any combination. Additional monitors may also be added without interrupting execution. PLEXIL does not insert any latency in execution by polling monitors in some pre-defined order.

C. Plan Execution and Quiescence

Execution in PLEXIL is driven by external events. The set of events includes the following:

- *Events related to LookupOnChange and LookupWithFrequency (meaning that the value of an external state variable has changed)*
- *Received acknowledgement of a command*
- *Received the output of a command, etc*

Time is acquired from the external system (via lookups). The execution of a plan proceeds in discrete time steps, called *macro-steps*. All external events are processed in the order in which they are received. An external event and all its cascading effects are processed before the next event is processed. A macro-step of execution consists of a number of *micro-steps* that process cascading effects.

D. Micro Steps

Micro-steps correspond to transitions that modify *only the local data* in the executive, i.e. node states and outcomes and the values of the local variables. A micro-step is defined as the synchronous parallel execution of a transition, as defined by the state transition diagrams (see Appendix A). Priorities are used for solving conflicting parallel assignments (e.g. assignment to the same variables). Precedence order on condition changes is used to resolve conflicts when multiple condition changes occur simultaneously.

Quiescence:

Quiescence is the repeated application of micro-steps until no more transitions are enabled. Given an external event, all the nodes waiting for that event are transitioned in parallel to their next states. If these transitions trigger other condition changes (due to changes in node state values) these are executed until there are no more enabled transitions, i.e. until quiescence. At this point, the next external event from the queue is handled.

Macro Steps and Plan Execution:

The execution of a plan starts in an initial state: the root node of the plan is in state *Waiting*. All the other nodes are in state *Inactive*. The execution of the plan starts by first evaluating all the gate condition for state *Waiting* in the root. The execution of a macro-step proceeds as follows:

- A cycle of quiescence is performed, by a repeated application of micro-steps until no transitions are enabled. The micro-steps may modify only the local data of the executive (i.e. the values of PLEXIL variables, node states and outcomes, etc.)
- The execution waits for an external event; when a new event is received, the gate conditions associated with the node states are evaluated (some of them may become enabled) and a new cycle of quiescence is performed.

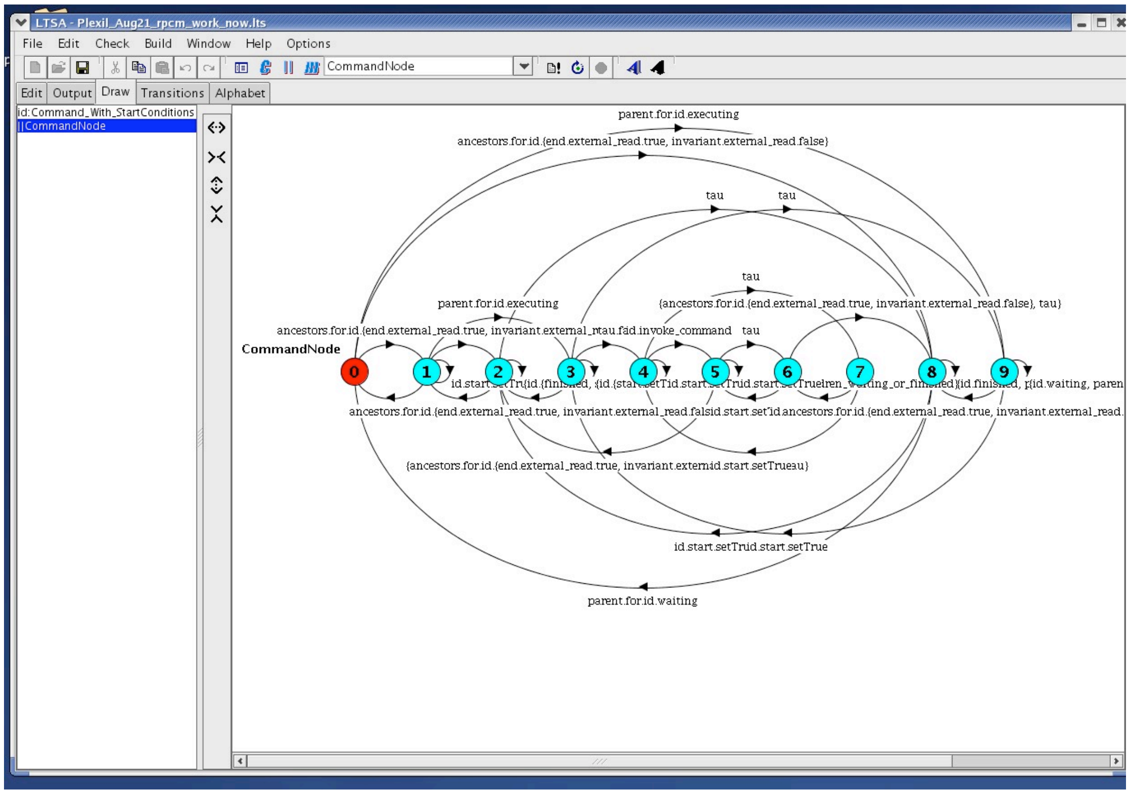

Figure 5. The LTSA tool.

V. Verification and Validation

Thorough testing and software certification is mandatory for any space application software, in particular for autonomous software technology. However, traditional methods for testing and certification are not sufficient for certifying autonomy software, due to the subtle interactions of in-situ reasoning with unknown environments. Formal verification on the other hand, uses analytical methods to prove that a system conforms to its requirements. Formal verification checks universal properties and it examines all the behaviors of a system, independent of a particular set of environment inputs. Verification technologies that have been applied to PLEXIL are discussed below.

A. LTSA Tool

The Labeled Transition System Analyzer (LTSA)[2] is a tool that supports *automated formal verification* of software components modeled as finite state machines. Components are assembled together using a classic automata product construction where shared actions are synchronized and remaining actions are interleaved. The tool can check absence of deadlocks, assertions and more general properties that encode system requirements. The properties can be written as finite state machines or (fluent) linear temporal logic). The input language "FSP" (<u>F</u>inite <u>S</u>tate <u>P</u>rocesses) of the tool is a process-algebra style notation that contains special operators for hiding and renaming of actions. The LTSA tool also features graphical display of LTSs, interactive simulation and graphical animation of behavior models, all helpful aids in both design and verification of system models.

B. Verification of PLEXIL state transition diagrams

The PLEXIL state transition diagrams were encoded into FSP (~200 lines of code). The encoding is straightforward: there is a close correspondence between the sate transition diagrams and the finite state machines that can be described in LTSA. A screen-shot of the LTSA tool is shown in Figure 5. In the following we show a (simplified) FSP code for encoding the transitions from state WAITING. A diagrammatic representation of the transitions from state WAITING are shown in Figure 15.

```
   Plexil(N=List)        =        (…        ->
INACTIVE[N][No_Fail]),
   INACTIVE[n:NodeType][o:OutcomeType] = …
,
   WAITING[n:NodeType][o:OutcomeType] =
        (start.read[‘true] ->
             (pre.read[‘true]              ->
EXECUTING[n][o]
             |
{pre.read[‘false],pre.read[‘unknown]}    ->
failure -> ITERATION_ENDED[n][Failure]
             )
        |    ancestor_end.read[‘true]    ->
skipped -> FINISHED[n][Skipped]
        | ancestor_invariant.read[‘false] -> skipped -> FINISHED[n][Skipped]
        ),
   …
   FINISHED[n:NodeType][o:OutcomeType] = END.
   || ListNode = Plexil(List).
   || CommandNode = Plexil(Command).
```

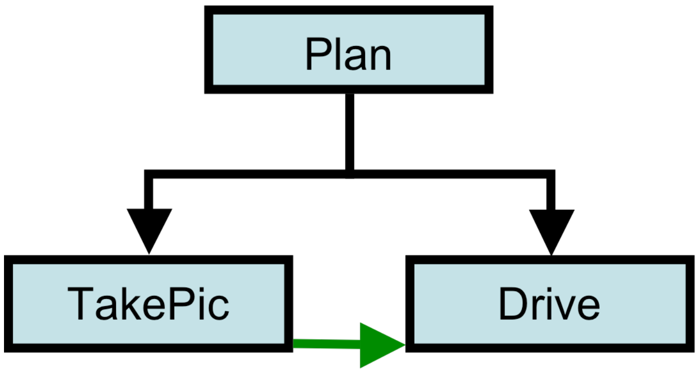

Figure 6. Drive starts when TakePicture finishes.

The Plexil model (which is parameterized by the node and its outcome) contains all the node states, e.g. INACTIVE, WAITING, etc. Transitions are specified with "`->`" and different choices are encoded with "`|`". We introduced special read actions for checking node conditions; for example action "`start.read[‘true]`" models that the node start condition evaluates to true.

After encoding the model, we simulated it and checked for several properties: deadlock freedom, assertions and LTL properties. It is interesting to note that the properties that we checked were expected to hold for the transition diagrams, and hence, for any possible node (irrespective of a particular PLEXIL plan).

Property: In state EXECUTING the node outcome can not be Failure.

This property was encoded as an assertion. Note that this property should hold for any node. For an early version of the node state transition diagrams, LTSA reported a counterexample showing that the property does not hold in looping nodes. The counterexample shows a scenario where a looping node fails in the first loop iteration (due to a pre-condition failure), the node outcome is set to failure, the node starts a new iteration and transitions to state EXECUTING (with the outcome still set to Failure). We corrected the diagrams and our model by adding explicit reset operations for the outcome values.

Property: It is always the case that if the ancestor invariant becomes false, then eventually the node goes to state FINISHED.

This property was encoded as a linear temporal formula and it was checked successfully on the model
assert P = (ancestor_invariant.read.false ⇒ ◊-> <> enter_state_finished)

A number of other properties were checked and these helped detect infrequently occurring subtle problems in the state transition diagrams that would not have easily been caught with testing alone.

C. Verification of PLEXIL plans

As mentioned, the LTSA models for the semantics diagrams allowed us to check generic properties for all the possible nodes (and plans). In order to check properties of specific plans, we defined an automatic translation from PLEXIL plans into FSP. Note that the LTSA model that encodes the semantics diagrams defines a "template" which is the same for any plan. For a particular plan, we generate particular *instances* from this template, and the corresponding renaming for each node in the plan. The renamings encode parent-child relationships and also node dependencies. For example, here are the template instances that are generated for the plan illustrated in Figure 6.

```
|| ExamplePlan = (List(Plan) || Command(TakePic) || Command (Drive)).
/* parent child relationships */

{TakePic,Drive}/Plan.children
Plan/Drive.parent
Plan/TakePic.parent
/* dependencies */

Drive.start.setTrue/TakePic.finished
```

The plan consists of three template instances (one list called "Plan" and two commands "TakePic" and "Drive") that run in parallel. The renamings ensure proper synchronization between these instances: by setting the children of the "Plan" instance to be "TakePic" and "Drive" and the parent for both "Drive" and "TakePic" to be "Plan". The remaining constraint specifies that the start condition of "Drive" is set to true when the "TakePic" instance is in state "finished". Such encodings allow us to check plans for deadlock freedom (i.e. to discover circular node dependencies), infinite loops on internal actions (to ensure termination of quiescence) and conformance between translated plans and specific plan requirements. Note that this check is done at an abstract level (we do not model time explicitly, local variables are also abstracted away); we will need to investigate more powerful verification tools (such as UPAAL) to handle these issues.

VI. Universal Executive

The Universal Executive (UE) is a light-weight and efficient execution system that executes PLEXIL plans with the defined PLEXIL semantics. The executive is coded in C++ and has been tested to build on VxWorks, Unix variants, and Linux. All the demonstration and tests described in this paper were performed using the UE. The UE also has a lightweight simulation script for testing purposes.

A. Lightweight

The research version of the UE (for which no explicit effort to optimize space and CPU usage has been made) is currently between 0.973-1.5 MB depending on the operating system. This includes the XML parser and utility libraries. In terms of runtime memory statistics, to run the SCOUT rover inspection scenario described in the next section, it takes:

1) At the most 400k of heap + stack. The node tree, expressions, executive, and the simulated world (which is very inefficient in memory usage and has not been optimized since it will not be part of a deployed system) for this scenario can be contained in 360k of memory.
2) Execution and simulation of the inspection scenario takes 0.055 seconds.

The PLEXIL plan for the SCOUT inspection scenario consists of 36 nodes, 14 of which are command nodes, 15 assignment nodes, and one update node. It is possible to garbage collect completed plan fragments, so the plan in memory need not grow with mission time.

B. Efficient

In the UE, expressions are evaluated completely only when initially relevant, and only re-evaluated to the degree necessitated by changes to their leaves.

In addition complex computations may be done outside the UE through function call nodes, allowing them to be scheduled towards some optimum.

The UE state cache makes it is possible to send the full text of a state and its parameters only once and use an "id" thereafter to conserve bandwidth. Currently this is not done for commands and functions, but could easily be expanded. Doing this essentially reduces arbitrary sized messages to ~8 bytes.

C. Easy to implement interface

Interfacing the UE to different control layer software is fairly easy. There is a well defined external interface that can connect simultaneously to a number of decision-support systems, control layer modules, graphical user interfaces, etc. PLEXIL has update nodes that define what information must be passed back to the planner interface. In addition the UE also publishes execution status during execution, which includes all node state transitions, outcomes, and bindings of local variables among other things. It is fairly easy to be aware of the current state of the executive and of plan execution when the UE is executing PLEXIL plans. It is also fairly easy to append plan fragments while the UE is executing. It is also easy to immediately abort or "cleanly end" any PLEXIL plan that the UE is executing.

VII. Demonstration of use

A few examples of the scenarios that PLEXIL may be applied to were described in section II. Here we present two applications where PLEXIL and the universal executive are currently being used.

A. Robotic inspection of Lunar rover prototype

This scenario targets habitats and vehicles for Lunar surface exploration, where humans and robots work together. This scenario is part of a demonstration involving multiple NASA centers. At the moment tests are being performed at individual NASA centers. The demonstration of this scenario will be held in September 2006 at Meteor Crater in Arizona with all the rovers, astronauts, and habitats present at one location. The test will demonstrate operations that occur after crew returns from a planetary EVA (extra vehicular activity) sortie on the SCOUT[3] (Science Crew Operations and Utility Testbed) rover to a Lunar base site. The SCOUT rover is capable of transporting two astronauts and navigating rough terrain. The following three robots are stationed at the base site to support post lunar sortie operations:

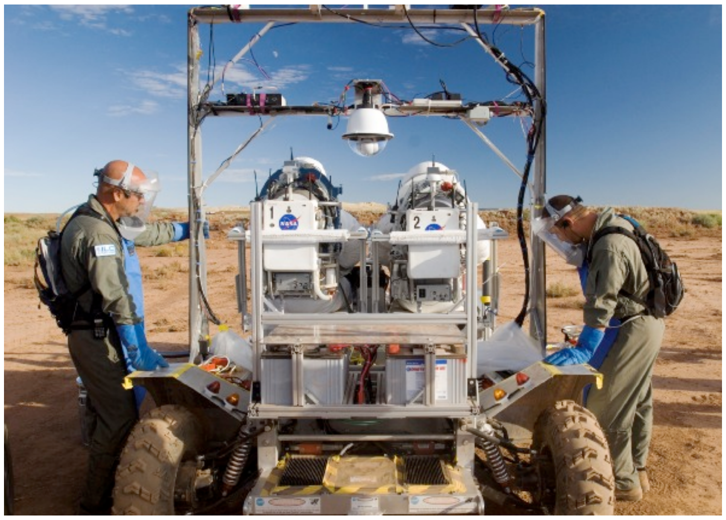
Figure 7. SCOUT rover

1) ATHLETE (All-Terrain Hex-Legged Extra-Terrestrial Explorer) rover from Jet Propulsion Lab, which is capable of "rolling" and "walking" over rough Lunar terrain. In this scenario ATHELETE will be equipped with a modular pressurized crew habitat.
2) Robonaut[4], which is a humanoid robot designed by NASA Johnson Space Center. In this scenario Robonaut will be in its mobile Centaur configuration.
3) K10[5] rover from NASA Ames Research Center, which is a light-weight mobile robot. In this scenario K10 is the inspection robot. PLEXIL and Universal Executive are currently implementing the control, monitoring, and exception handling for the K10 rover.

There are a number of operations performed in this scenario. The EVA crew returns to the "Lunar" base site from after a field excursion. The crew dismount SCOUT and mount the ATHELETE rover to return to the habitat. Robonaut approaches the SCOUT rover and unloads payload from it. After Robonaut completes its task, K10 approaches SCOUT and performs a visual inspection of SCOUT. In order to complete the inspection K10 runs the Universal Executive on-board, which executes a manually created PLEXIL plan to:

1) Autonomously drive K10 from its current location to SCOUT.
2) Circumnavigate Scout, stopping at 4-10 inspection points where four points are the primary inspection points and must be inspected and the rest are secondary and may be skipped if time is running out or a

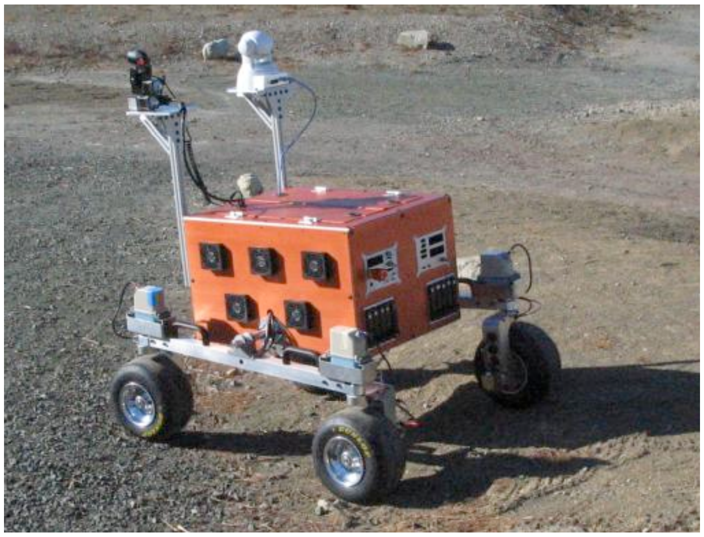
Figure 8. K10 rover.

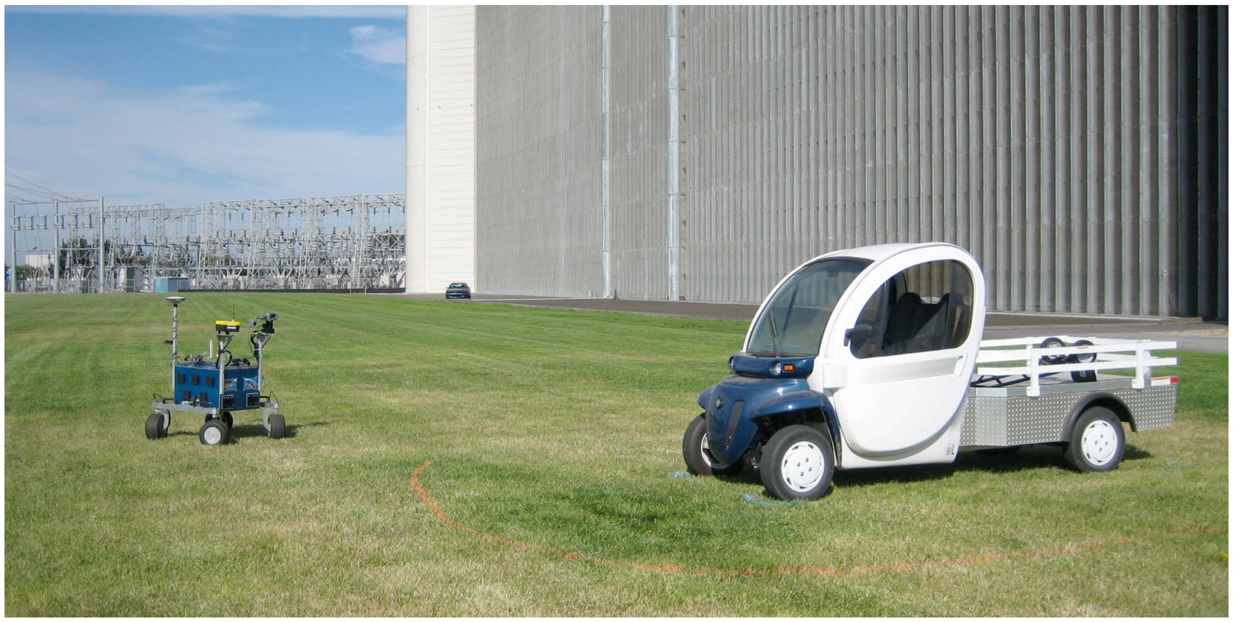
Figure 9. Testing at NASA Ames. A K10 is performing a visual inspection of the electric car (being used to simulate SCOUT) to prepare for a test in Meteor crater.

soft failure occurs. Soft failures are failures that K10 can recover form autonomously, hard failures are failures that require human intervention.

3) At each inspection point K10 collects panoramic images and sends them to operations control in the "habitat" via WiFi. Crew or ground control then performs visual inspection by analyzing the resulting images.

The PLEXIL plan for the inspection task includes a lot of contingency detection and handling. Even before the field test a number of contingencies have occurred during testing, which the Universal Executive running the PLEXIL plan was able to detect. These include communications failures, skipping secondary node because a inspection point took longer than expected, detection of stuck wheel, where a wheel was temporarily stuck and the locomotion system (which reads the encoders) thought the wheels was still moving. We expect to have more performance statistics from the upcoming demonstration in Meteor crater.

B. Power system management on International Space Station (ISS)

This scenario is performed in the context of the Spacecraft Autonomy for Spacecraft and Habitats (SAVH) project. The overall SAVH architecture is shown in Figure 10. One of the goals of this project is to increase crew autonomy and capability to manage vehicle and systems. Toward this end the power system management scenario demonstrates closed-loop planning and execution for power system management. For this demonstration, International Space Station flight software is used to simulate a similar interface for the future Crew Exploration Vehicle (CEV). A decision support system creates a plan to manage the power system. This plan is then translated from the planner representation to PLEXIL and sent over the transport layer to the Universal Executive (UE). The UE executes this plan by executing Commands and reading telemetry from an interface to the ISS simulator.

Execution on the ISS is performed by executing Program Unique Identifiers or PUIs. The UE sends updates on execution status and the state of the ISS to the decision support system via the transport layer which closes the control loop. The CASPER (Continuous Activity Scheduling, Planning Execution and Replanning [19]) planning system is used as the decision-support system for this scenario.

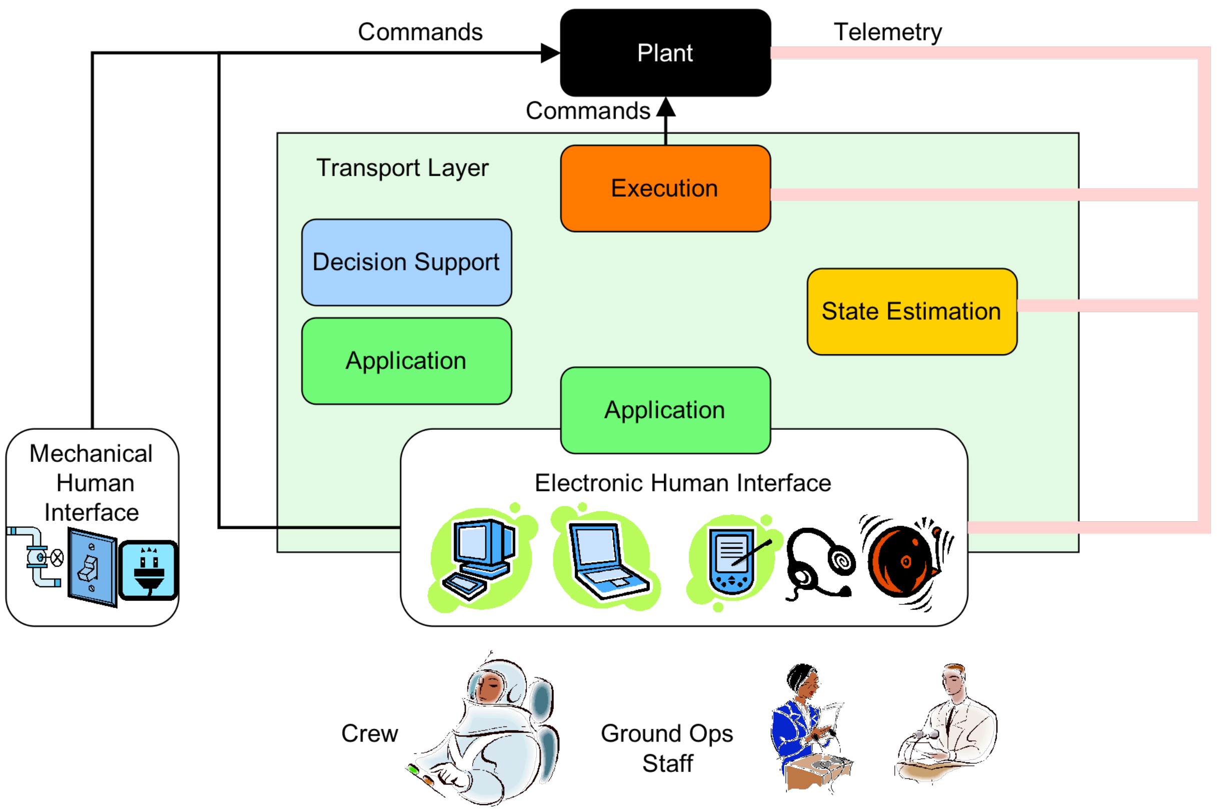

Figure 10. Components available in the Spacecraft Autonomy for Vehicles and Habitats scenario.

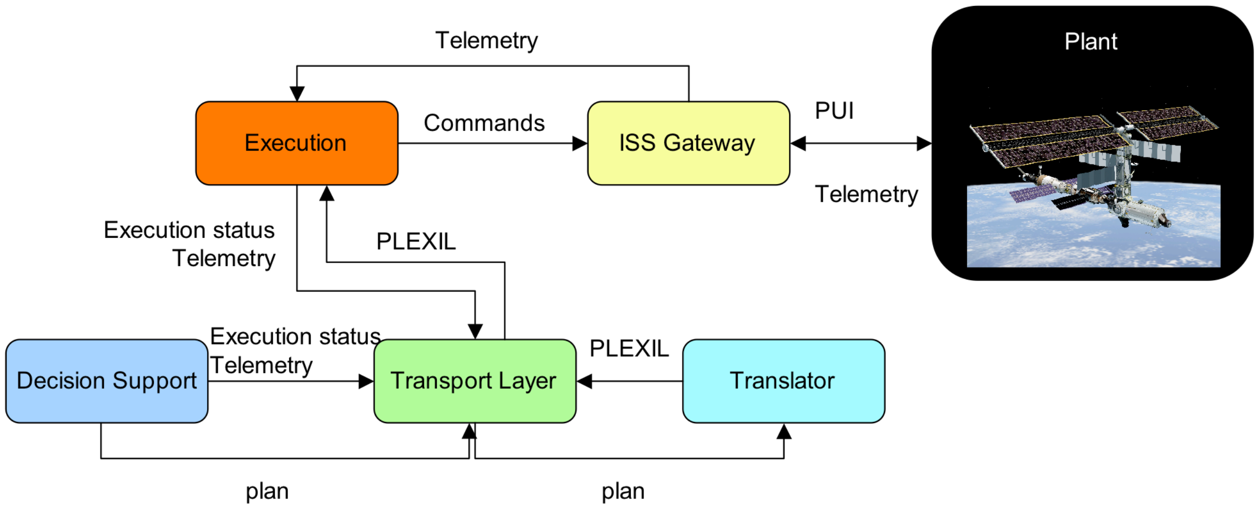

Figure 11. Autonomous power system management scenario for ISS.

VIII. Conclusion

PLEXIL provides a well-defined foundation on which further capabilities can be built. Extensions to provide resource level projections and utility trade-off specifications are straight-forward. Further extensions will include algorithms for simple scheduling rules that can be applied during run-time to handle delayed decisions. Finally, connections to decision-making software provides safety checking, projections, and the ability to make on-line decisions for situations not covered by a given PLEXIL plan. In all these extensions, the objective will be to build on the well-defined semantics for PLEXIL and extend them to cover the expanded capabilities.

Appendix A : Node state transition diagrams

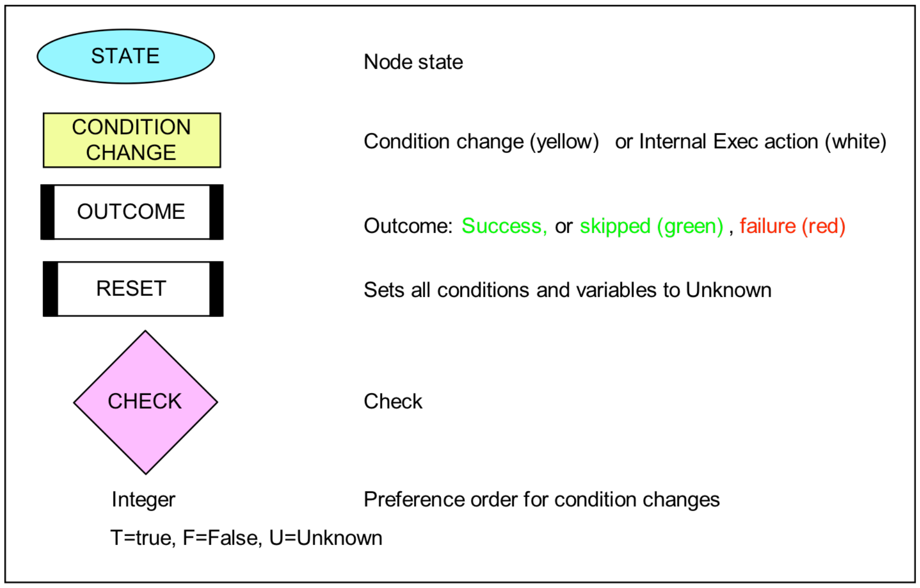
STATE
Node state
CONDITION CHANGE
Condition change (yellow) or Internal Exec action (white)
OUTCOME
Outcome: Success, or skipped (green) , failure (red)
RESET
Sets all conditions and variables to Unknown
CHECK
Check
Integer
Preference order for condition changes
T=true, F=False, U=Unknown

Figure 12. Legend for node state transition diagrams..

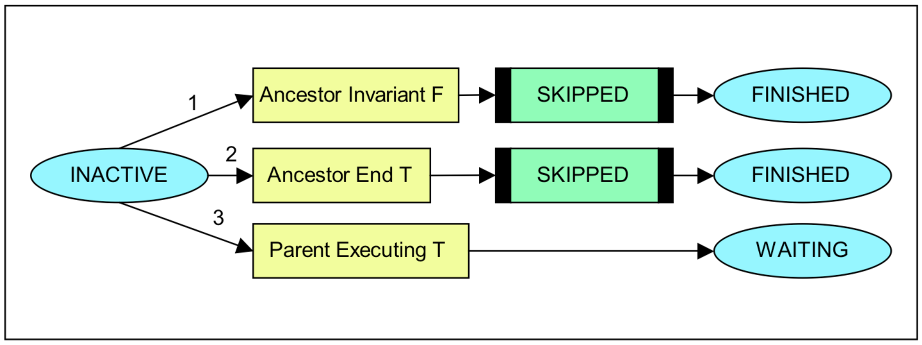

Figure 13. All transitions from state inactive for all node types.

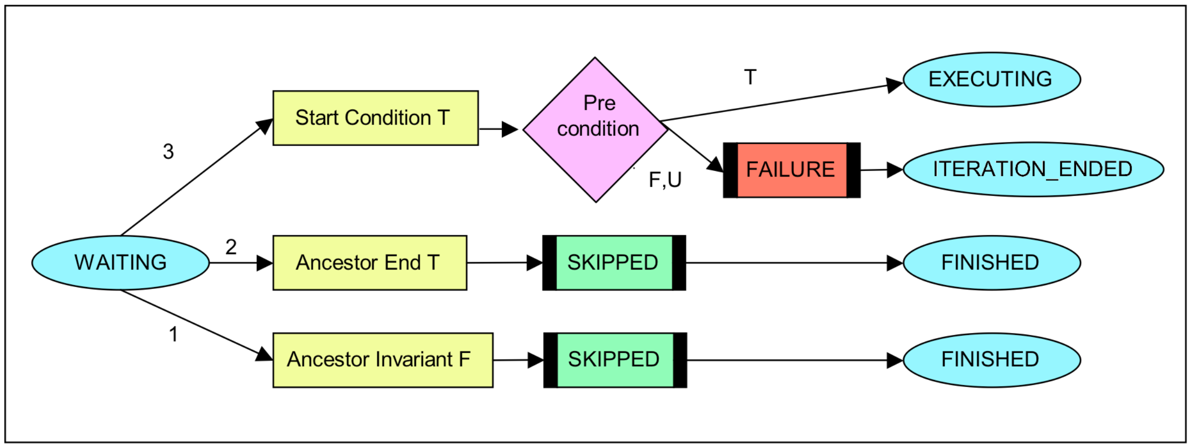

Figure 15. All transitions from state waiting for all node types.

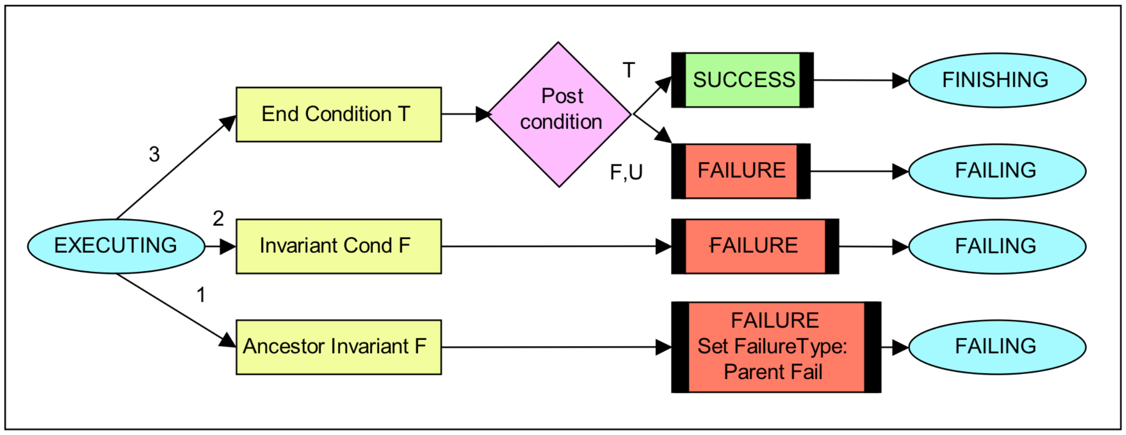

Figure 14. All transitions from state executing for NodeList nodes.

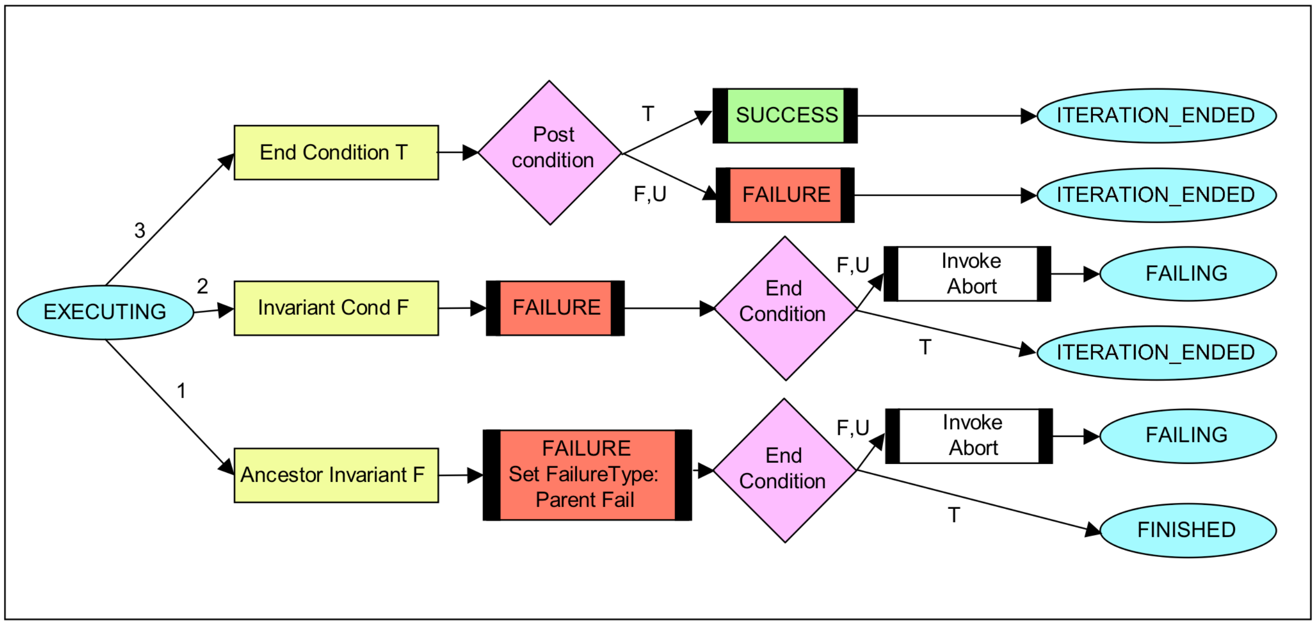

Figure 16. All transitions from state *executing* for *Command, PlanRequest*, and *Update* nodes.

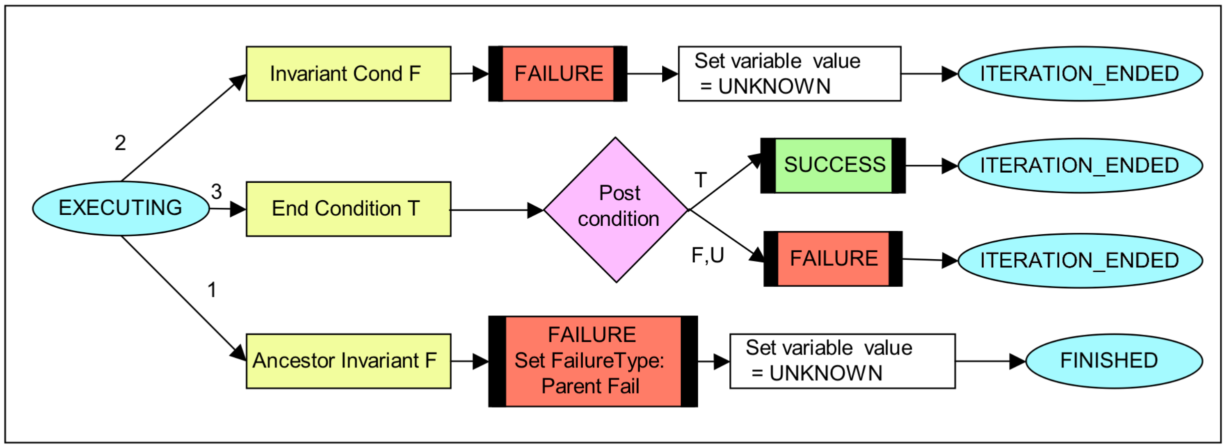

Figure 19. All transitions from state *executing* for simple *assignment* nodes.

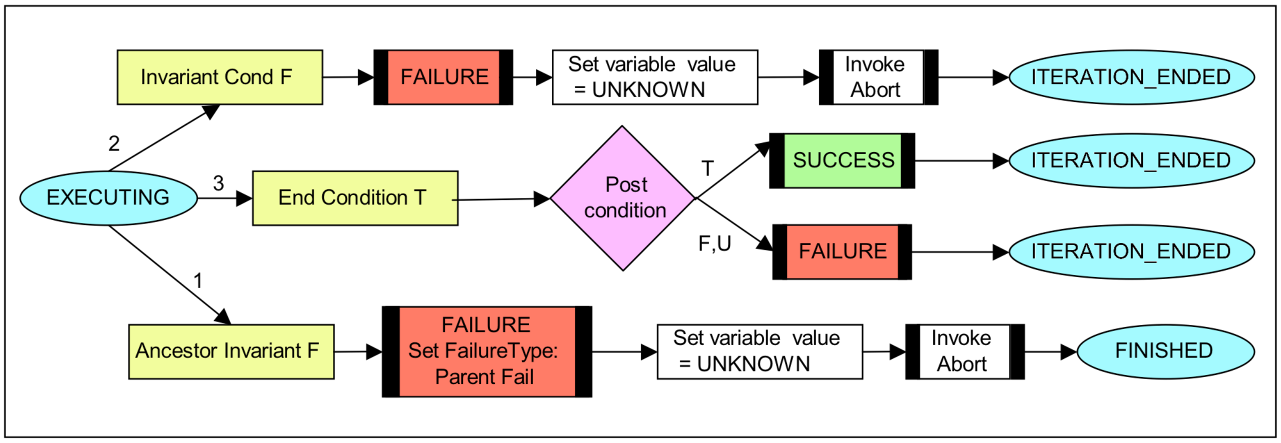

Figure 18. All transitions from state *executing* for *FunctionCall* nodes.

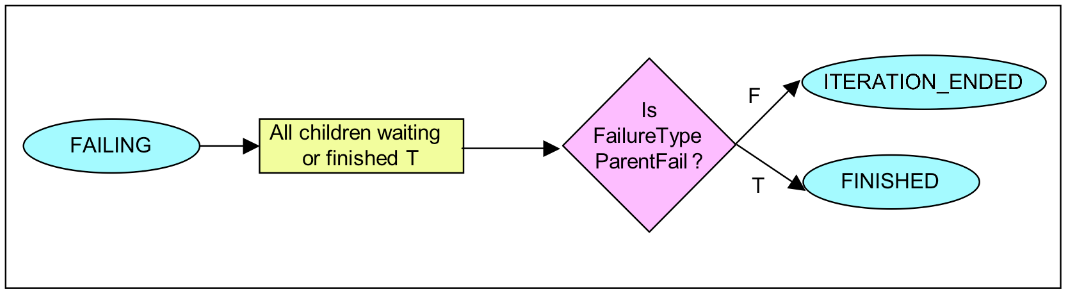
FAILING
All children waiting or finished T
Is FailureType ParentFail ?
F
ITERATION_ENDED
T
FINISHED

Figure 17. All transitions from state *failing* for *NodeList* nodes.

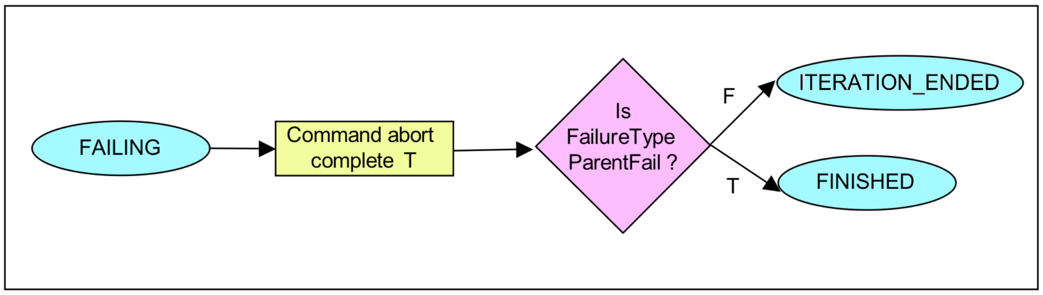

Figure 22. All transitions from state failing for Command, PlanRequest, and Update nodes.

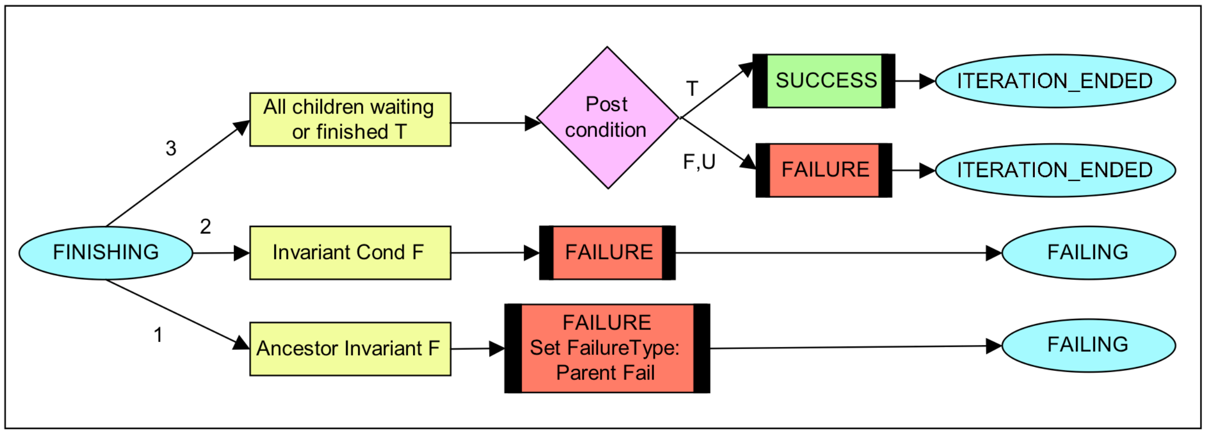

Figure 21. All transitions rfom state Finishing. Only NodeList nodes can be in state finishing.

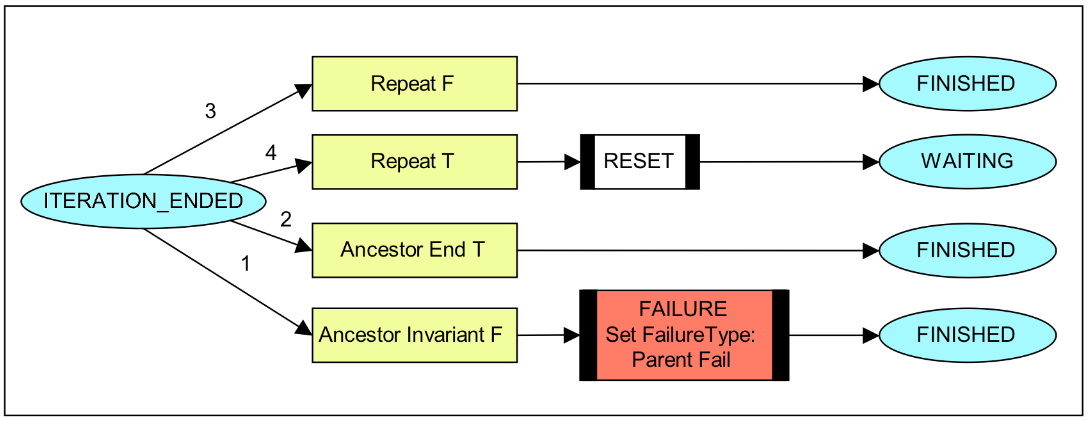

Figure 20. All transitions from state iteration ended for all repeating nodes. Any node type may be repeating.

Acknowledgments

Reid Simmons, Tara Estlin, and Richard Washington are part of the vision for PLEXIL and the universal executive. Cesar Munoz helped formalize PLEXIL semantics. Chuck Fry, Michael Dalal, Ron van Hoof, Gregg Rabideau, Steve Schaffer, David Kortenkamp, Pete Bonasso, Conor McGann, Chin Shea, and Debra Schreckenghost helped implement the International Space Station application. The robotic inspection scenario was designed as a joint effort by a number of robotics teams from JPL, JSC, and NASA Ames. Matt MacMahon and Hans Utz played a significant role in implementing the robotic inspection application. Mihaela Gheorghiu wrote an automatic translator from PLEXIL plans into the input language part of the LTSA verification tool. Dimitra Giannakopoulou helped with the LTSA verification. This work was supported by the Mars Technology and Exploration Technology Development Programs.

References

[1]Dowek, G., Munoz, C, Pasareanu C., The Semantics of PLEXIL (in preparation).

[2]Magee, J. and Kramer, J., *Concurrency: State Models & Java Programs*: John Wiley & Sons, 1999.

[3]Bortman, H., "Desert RATS Test Robotic Rover," *Astrobiology Magazine*, URL: http://www.astrobio.net/news/article1727.html [cited 13 August 2006].

[4] Ambrose, R. O., Aldridge, H., Askew, R.S., "NASA's ROBONAUT System", HURO 99, Tokyo Japan, October 1999.

[5]Terrence Fong, Jean Scholtz, Julie A. Shah, Lorenzo Fluckiger, Clayton Kunz, David Lees, John Schreiner, Michael Siegel, Laura M. Hiatt, Illah Nourbakhsh, Reid Simmons, Robert Ambrose, Robert Burridge, Brian Antonishek, Magda Bugajska, Alan Schultz and J. Gregory Trafton, "A Preliminary Study of Peer-to-Peer Human-Robot Interaction", *To appear in Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Taipei, October, 2006.

[6]S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau, "Integrated Planning and Execution for Autonomous Spacecraf" IEEE Aerospace Conference (IAC 1999). Aspen, CO. March 1999